企业级Hadoop平台二次开发-第五天

一、课前准备

1. 掌握之前的HDFS源码知识

二、课堂主题

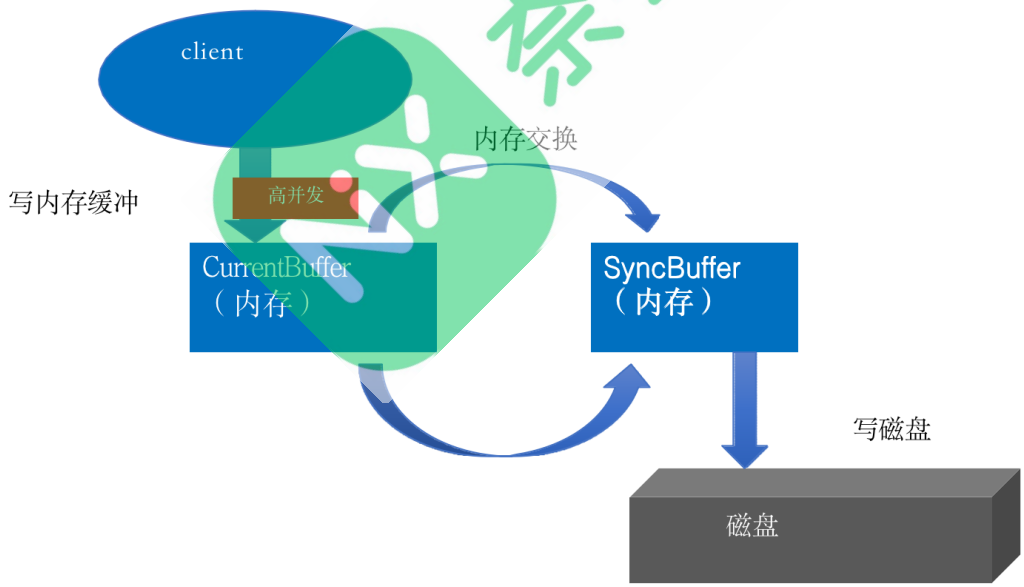
本次课对HDFS源码进行二次开发提升集群性能和稳定性，总结常用的设计模式

三、课程目标

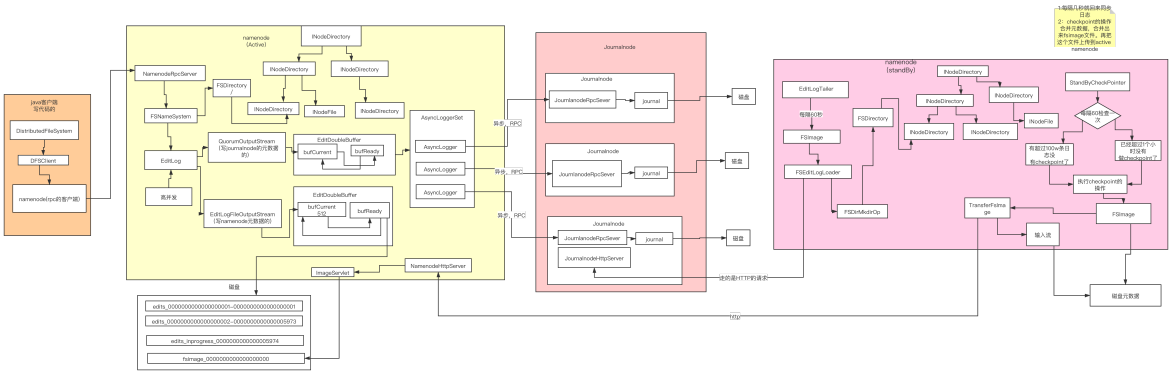
1. 解决NameNode瞬时高并发导致短暂不可用问题
2. 解决NameNode Full GC导致集群退出问题剖析
3. DataNode进行锁优化
4. 常见设计模式总结
5. 项目总结

四、知识要点

4.1 NameNode瞬时高并发导致短暂不可用



4.2 NameNode Full GC导致集群退出问题剖析



4.3 对DataNode进行锁优化

总体来说HDFS的锁是用得比较重的，有很多地方都值得优化，本次我们挑一个典型的地方进行锁的优化。

源码中观察一下BPOfferService类读写锁的使用。

```
  private final ReentrantReadWriteLock mReadWriteLock =new
ReentrantReadWriteLock();
  private final Lock mReadLock  = mReadWriteLock.readLock();
  private final Lock mWriteLock = mReadWriteLock.writeLock();
```

4.3.1 写锁使用：

（1）BPServiceActor获取到了NamespaceInfo，设置这个数据

```
void verifyAndSetNamespaceInfo(NamespaceInfo nsInfo) throws IOException {
    writeLock();
    try {
      //与第一个Namenode进行交互，返回来的信息
      //因为是跟第一个Namenode交互所以bpNSInfo==null
      if (this.bpNSInfo == null) {
        this.bpNSInfo = nsInfo;
        boolean success = false;

        // Now that we know the namespace ID, etc, we can pass this to the DN.
        // The DN can now initialize its local storage if we are the
        // first BP to handshake, etc.
        try {
          dn.initBlockPool(this);
          success = true;
        } finally {
          if (!success) {
            // The datanode failed to initialize the BP. We need to reset
            // the namespace info so that other BPService actors still have
            // a chance to set it, and re-initialize the datanode.
            this.bpNSInfo = null;
          }
        }
      } else {
        //代码走到这儿说明 是已经访问了第二个namenode了
        checkNSEquality(bpNSInfo.getBlockPoolID(), nsInfo.getBlockPoolID(),
            "Blockpool ID");
        checkNSEquality(bpNSInfo.getNamespaceID(), nsInfo.getNamespaceID(),
            "Namespace ID");
        checkNSEquality(bpNSInfo.getClusterID(), nsInfo.getClusterID(),
```

```
            "Cluster ID");
      }
    } finally {
      writeUnlock();
    }
  }
```

(2) BPServiceActor准备向NameNode注册的时候创建DatanodeRegistration数据

```
DatanodeRegistration createRegistration() {
    writeLock();
    try {
      Preconditions.checkState(bpNSInfo != null,
          "getRegistration() can only be called after initial handshake");
      return dn.createBPRegistration(bpNSInfo);
    } finally {
      writeUnlock();
    }
  }
```

(3) BPServiceActor注册成功了之后设置DatanodeRegistration

```
void registrationSucceeded(BPServiceActor bpServiceActor,
      DatanodeRegistration reg) throws IOException {
    writeLock();
    try {
      if (bpRegistration != null) {
        checkNSEquality(bpRegistration.getStorageInfo().getNamespaceID(),
            reg.getStorageInfo().getNamespaceID(), "namespace ID");
        checkNSEquality(bpRegistration.getStorageInfo().getClusterID(),
            reg.getStorageInfo().getClusterID(), "cluster ID");
      } else {
        bpRegistration = reg;
      }

      dn.bpRegistrationSucceeded(bpRegistration, getBlockPoolId());
      // Add the initial block token secret keys to the DN's secret manager.
      if (dn.isBlockTokenEnabled) {
        dn.blockPoolTokenSecretManager.addKeys(getBlockPoolId(),
            reg.getExportedKeys());
      }
    } finally {
      writeUnlock();
    }
  }
```

(4) 如果DataNode关闭，此时就会关闭某个BPServiceActor，此时会清空一些数据

```
void shutdownActor(BPServiceActor actor) {
    writeLock();
    try {
      if (bpServiceToActive == actor) {
        bpServiceToActive = null;
      }
```

```
      bpServices.remove(actor);

      if (bpServices.isEmpty()) {
        dn.shutdownBlockPool(this);
      }
    } finally {
      writeUnlock();
    }
  }
```

(5) 每隔3秒钟跟NameNode进行一次心跳，如果发现NameNode的active/standby状态发生了变化，此时就会更新一些数据

```
void updateActorStatesFromHeartbeat(
      BPServiceActor actor,
      NNHAStatusHeartbeat nnHaState) {
    writeLock();
    try {
      final long txid = nnHaState.getTxId();

      final boolean nnClaimsActive =
          nnHaState.getState() == HAServiceState.ACTIVE;
      final boolean bposThinksActive = bpServiceToActive == actor;
      final boolean isMoreRecentClaim = txid > lastActiveClaimTxId;

      if (nnClaimsActive && !bposThinksActive) {
        LOG.info("Namenode " + actor + " trying to claim ACTIVE state with " +
            "txid=" + txid);
        if (!isMoreRecentClaim) {
          // Split-brain scenario - an NN is trying to claim active
          // state when a different NN has already claimed it with a higher
          // txid.
          LOG.warn("NN " + actor + " tried to claim ACTIVE state at txid=" +
              txid + " but there was already a more recent claim at txid=" +
              lastActiveClaimTxId);
          return;
        } else {
          if (bpServiceToActive == null) {
            LOG.info("Acknowledging ACTIVE Namenode " + actor);
          } else {
            LOG.info("Namenode " + actor + " taking over ACTIVE state from " +
                bpServiceToActive + " at higher txid=" + txid);
          }
          bpServiceToActive = actor;
        }
      } else if (!nnClaimsActive && bposThinksActive) {
        LOG.info("Namenode " + actor + " relinquishing ACTIVE state with " +
            "txid=" + nnHaState.getTxId());
        bpServiceToActive = null;
      }

      if (bpServiceToActive == actor) {
        assert txid >= lastActiveClaimTxId;
        lastActiveClaimTxId = txid;
      }
    } finally {
      writeUnlock();
```

```
    }
  }
```

(6) 每次发送心跳给NameNode都有可能会带回来一些指令，比如说通知DataNode复制某个block副本到其他的DataNode上去

```
boolean processCommandFromActor(DatanodeCommand cmd,
      BPServiceActor actor) throws IOException {
    assert bpServices.contains(actor);
    if (cmd == null) {
      return true;
    }
    /*
     * Datanode Registration can be done asynchronously here. No need to hold
     * the lock. for more info refer HDFS-5014
     */
    if (DatanodeProtocol.DNA_REGISTER == cmd.getAction()) {
      // namenode requested a registration - at start or if NN lost contact
      // Just logging the claiming state is OK here instead of checking the
      // actor state by obtaining the lock
      LOG.info("DatanodeCommand action : DNA_REGISTER from " + actor.nnAddr
          + " with " + actor.state + " state");
      actor.reRegister();
      return false;
    }
    writeLock();
    try {
      if (actor == bpServiceToActive) {
        return processCommandFromActive(cmd, actor);
      } else {
        return processCommandFromStandby(cmd, actor);
      }
    } finally {
      writeUnlock();
    }
  }
```

4.3.2 读锁使用：

（1）从BPOfferService中管理的一个NamespaceInfo数据里提取出来对应的NameNode的blockPoolID这个核心的id数据。

```
 String getBlockPoolId() {
    readLock();
    try {
      if (bpNSInfo != null) {
        return bpNSInfo.getBlockPoolID();
      } else {
        LOG.warn("Block pool ID needed, but service not yet registered with NN",
            new Exception("trace"));
        return null;
      }
    } finally {
      readUnlock();
    }
  }
```

（2）获取核心的NamespaceInfo数据

```
  NamespaceInfo getNamespaceInfo() {
    readLock();
    try {
      return bpNSInfo;
    } finally {
      readUnlock();
    }
  }
```

（3）获取Active NameNode对应的rpc接口代理

```
  DatanodeProtocolClientSideTranslatorPB getActiveNN() {
    readLock();
    try {
      if (bpServiceToActive != null) {
        return bpServiceToActive.bpNamenode;
      } else {
        return null;
      }
    } finally {
      readUnlock();
    }
  }
```

我们整体看了读锁和写锁加的所有位置。

我们注意观察读锁里有一个方法getBlockPoolId()，这个方法是HDFS里使用频率很高的一个方法。在这几个地方都用了这个这个方法：

BPServiceActor里面的方法：

（1）每隔3秒心跳的方法

```
HeartbeatResponse sendHeartBeat() throws IOException {
    StorageReport[] reports =
        dn.getFSDataset().getStorageReports(bpos.getBlockPoolId());
    if (LOG.isDebugEnabled()) {
      LOG.debug("Sending heartbeat with " + reports.length +
                " storage reports from service actor: " + this);
    }

    VolumeFailureSummary volumeFailureSummary = dn.getFSDataset()
        .getVolumeFailureSummary();
    int numFailedVolumes = volumeFailureSummary != null ?
        volumeFailureSummary.getFailedStorageLocations().length : 0;
        //TODO 发送心跳
       //获取到NameNode的代理，发送心跳
    return bpNamenode.sendHeartbeat(bpRegistration,
        reports,
        dn.getFSDataset().getCacheCapacity(),
        dn.getFSDataset().getCacheUsed(),
        dn.getXmitsInProgress(),
        dn.getXceiverCount(),
```

```
        numFailedVolumes,
        volumeFailureSummary);
  }
```

(2) 全量块汇报

```
 List<DatanodeCommand> blockReport() throws IOException {
    ......
    Map<DatanodeStorage, BlockListAsLongs> perVolumeBlockLists =
        dn.getFSDataset().getBlockReports(bpos.getBlockPoolId());

    // Convert the reports to the format expected by the NN.
    int i = 0;
    int totalBlockCount = 0;
    StorageBlockReport reports[] =
        new StorageBlockReport[perVolumeBlockLists.size()];

    for(Map.Entry<DatanodeStorage, BlockListAsLongs> kvPair :
perVolumeBlockLists.entrySet()) {
      BlockListAsLongs blockList = kvPair.getValue();
      reports[i++] = new StorageBlockReport(kvPair.getKey(), blockList);
      totalBlockCount += blockList.getNumberOfBlocks();
    }

    // Send the reports to the NN.
    int numReportsSent = 0;
    int numRPCs = 0;
    boolean success = false;
    long brSendStartTime = monotonicNow();
    long reportId = generateUniqueBlockReportId();
    try {
      if (totalBlockCount < dnConf.blockReportSplitThreshold) {
        // Below split threshold, send all reports in a single message.
        DatanodeCommand cmd = bpNamenode.blockReport(
            bpRegistration, bpos.getBlockPoolId(), reports,
              new BlockReportContext(1, 0, reportId));
        numRPCs = 1;
        numReportsSent = reports.length;
        if (cmd != null) {
          cmds.add(cmd);
        }
      } else {
        // Send one block report per message.
        for (int r = 0; r < reports.length; r++) {
          StorageBlockReport singleReport[] = { reports[r] };
          DatanodeCommand cmd = bpNamenode.blockReport(
              bpRegistration, bpos.getBlockPoolId(), singleReport,
              new BlockReportContext(reports.length, r, reportId));
          numReportsSent++;
          numRPCs++;
          if (cmd != null) {
            cmds.add(cmd);
          }
        }
      }
```

```
      success = true;
    } finally {
     ....
    }
    scheduleNextBlockReport(startTime);
    return cmds.size() == 0 ? null : cmds;
  }
```

上面心跳的方法是一个高频的操作，也就是说心跳的方法会调用getBlockPoolId方法，也就是说会因为调用getBlockPoolId方法而高频的加了读锁。

接下来我们再证明一下，HDFS 写锁也是一个高频的操作。

```
boolean processCommand(DatanodeCommand[] cmds) {
    if (cmds != null) {
      for (DatanodeCommand cmd : cmds) {
        try {
          if (bpos.processCommandFromActor(cmd, this) == false) {
            return false;
          }
        } catch (IOException ioe) {
          LOG.warn("Error processing datanode Command", ioe);
        }
      }
    }
    return true;
  }
```

处理这些指令的时候就是加了写锁：

```
boolean processCommandFromActor(DatanodeCommand cmd,
      BPServiceActor actor) throws IOException {
    assert bpServices.contains(actor);
    if (cmd == null) {
      return true;
    }
    /*
     * Datanode Registration can be done asynchronously here. No need to hold
     * the lock. for more info refer HDFS-5014
     */
    if (DatanodeProtocol.DNA_REGISTER == cmd.getAction()) {
      // namenode requested a registration - at start or if NN lost contact
      // Just logging the claiming state is OK here instead of checking the
      // actor state by obtaining the lock
      LOG.info("DatanodeCommand action : DNA_REGISTER from " + actor.nnAddr
          + " with " + actor.state + " state");
      actor.reRegister();
      return false;
    }
    writeLock();
    try {
      if (actor == bpServiceToActive) {
        return processCommandFromActive(cmd, actor);
      } else {
        return processCommandFromStandby(cmd, actor);
      }
```

```
    } finally {
      writeUnlock();
    }
  }
```

源码优化：

BPService里，源码修改如下：

```
volatile String blockPoolID;

void verifyAndSetNamespaceInfo(NamespaceInfo nsInfo) throws IOException {
   writeLock();
   try {
     //与第一个Namenode进行交互，返回来的信息
     //因为是跟第一个Namenode交互所以bpNSInfo==null
     if (this.bpNSInfo == null) {
       this.bpNSInfo = nsInfo;
       boolean success = false;

       // Now that we know the namespace ID, etc, we can pass this to the DN.
       // The DN can now initialize its local storage if we are the
       // first BP to handshake, etc.
       try {
         dn.initBlockPool(this);
         success = true;
       } finally {
         if (!success) {
           // The datanode failed to initialize the BP. We need to reset
           // the namespace info so that other BPService actors still have
           // a chance to set it, and re-initialize the datanode.
           this.bpNSInfo = null;
         }
       }
     } else {
       //代码走到这儿说明 是已经访问了第二个namenode了
       checkNSEquality(bpNSInfo.getBlockPoolID(), nsInfo.getBlockPoolID(),
           "Blockpool ID");
       checkNSEquality(bpNSInfo.getNamespaceID(), nsInfo.getNamespaceID(),
           "Namespace ID");
       checkNSEquality(bpNSInfo.getClusterID(), nsInfo.getClusterID(),
           "Cluster ID");
       //因为在这个时候getBlockPoolID 肯定是有值了
       if(blockPoolID == null){
         this.blockPoolID = bpNSInfo.getBlockPoolID();
       }
     }
   } finally {
     writeUnlock();
   }
 }
```

getBlockPoolId方法优化如下：

```
String getBlockPoolId() {
    //优化
    //优化
```

```
    if(this.blockPoolID != null){
      return this.blockPoolID;
    }

    readLock();
    try {
      if (bpNSInfo != null) {
        return bpNSInfo.getBlockPoolID();
      } else {
        LOG.warn("Block pool ID needed, but service not yet registered with NN",
            new Exception("trace"));
        return null;
      }
    } finally {
      readUnlock();
    }
  }
```

4.4 HDFS源码设计模式总结

4.4.1 命令设计模式

说明：HDFS源码的 offerService() 方法中。DataNode向Namenode进行心跳，而如果Namenode有什么事需要DataNode完成，就通过发送指令。

如果不用设计模式，代码如下：

```
public class WithoutCommandPatterDemo {

    public static void main(String[] args) {
       // Uitls utils= new Uitls();
        int sytle=1;
        if( 1 == sytle){
           Utils.read();
        }
        if( 2 == sytle){
            Utils.write();
        }
    }


    public static class Utils{
        public static void read(){
            System.out.println("读请求");
        }
        public static void write(){
            System.out.println("写请求");
        }

    }

}
```

用了设计模式

```
```

```
public class CommandPatterDemo {

    public static void main(String[] args) {
        if(true) {
            Context context = new Context(new ReadCommand());
            context.execute();
        }
    }



    public interface Command{
        void execute();
    }

    public static class ReadCommand implements  Command{
        public void execute() {
            System.out.println("读请求");
        }
    }

    public static  class WriteCommand implements  Command{
        public void execute() {
            System.out.println("写请求");
        }
    }

    public static class Context{
        private Command command;

        public Context(Command command) {
            this.command = command;
        }

        public void execute(){
            this.command.execute();
        }
    }


}
```

特点总结:

命令（Command）模式的定义如下：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。这样两者之间通过命令对象进行沟通，这样方便将命令对象进行储存、传递、调用、增加与管理。

命令模式的主要优点如下。
降低系统的耦合度。命令模式能将调用操作的对象与实现该操作的对象解耦。
增加或删除命令非常方便。采用命令模式增加与删除命令不会影响其他类，它满足“开闭原则”，对扩展比较灵活。
可以实现宏命令。命令模式可以与组合模式结合，将多个命令装配成一个组合命令，即宏命令。
方便实现 Undo 和 Redo 操作。命令模式可以与后面介绍的备忘录模式结合，实现命令的撤销与恢复。

其缺点是：可能产生大量具体命令类。因为计对每一个具体操作都需要设计一个具体命令类，这将增加系统的复杂性。

4.2.2 构建者模式

说明：Hadoop的RPC的服务端对象的创建就是用的构建者模式

如果没有用设计模式，代码如下：

```
public class WithoutPatternDemo {

    public static void main(String[] args) {
        Student student = new Student();
        if(true){
            System.out.println("非常复杂逻辑处理");
            student.setField1("1");
        }
        if(true){
            System.out.println("非常复杂逻辑处理");
            student.setField2("2");
        }
        if(true){
            System.out.println("复杂");
            student.setField3("3");
        }

        System.out.println(student);

    }


    public static class Student{
        private String field1;
        private String field2;
        private String field3;

        public String getField1() {
            return field1;
        }

        public void setField1(String field1) {
            this.field1 = field1;
        }

        public String getField2() {
            return field2;
        }
```

```
        public void setField2(String field2) {
            this.field2 = field2;
        }

        public String getField3() {
            return field3;
        }

        public void setField3(String field3) {
            this.field3 = field3;
        }

        @Override
        public String toString() {
            return "Student{" +
                    "field1='" + field1 + '\'' +
                    ", field2='" + field2 + '\'' +
                    ", field3='" + field3 + '\'' +
                    '}';
        }
    }
}
```

使用了设计模式代码如下：

```
public class BuidlerPattern {

    public static void main(String[] args) {
        Student student = new ConCreateStudent()
        .setField1("test1")
        .setField2("test2")
        .setField3("test3")
        .build();

        System.out.println(student);
    }

    public static class Student{
        private String field1;
        private String field2;
        private String field3;
        public String getField1() {
            return field1;
        }
        public void setField1(String field1) {
            this.field1 = field1;
        }
        public String getField2() {
            return field2;
        }
        public void setField2(String field2) {
            this.field2 = field2;
        }
```

```
        public String getField3() {
            return field3;
        }
        public void setField3(String field3) {
            this.field3 = field3;
        }
        @Override
        public String toString() {
            return "Student [field1=" + field1 + ", field2=" + field2 + ",
field3=" + field3 + "]";
        }
    }

    public interface Buidler{
        Buidler  setField1(String fields1);
        Buidler  setField2(String fields2);
        Buidler  setField3(String fields3);
        Student build();
    }
    public static class ConCreateStudent implements Buidler {
        Student student=new Student();

        @Override
        public Buidler setField1(String fields1) {
            System.out.println("复杂逻辑");
            student.setField1(fields1);
            return this;
        }

        @Override
        public Buidler setField2(String fields2) {
            System.out.println("复杂逻辑");
            student.setField2(fields2);
            return this;
        }

        @Override
        public Buidler setField3(String fields3) {
            System.out.println("复杂逻辑");
            student.setField3(fields3);
            return this;
        }

        @Override
        public Student build() {
            return student;
        }

    }

}
```

总结:

建造者（Builder）模式的定义：指将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示，这样的设计模式被称为建造者模式。它是将一个复杂的对象分解为多个简单的对象，然后一步一步构建而成。它将变与不变相分离，即产品的组成部分是不变的，但每一部分是可以灵活选择的。

该模式的主要优点如下：
各个具体的建造者相互独立，有利于系统的扩展。
客户端不必知道产品内部组成的细节，便于控制细节风险。

其缺点如下：
产品的组成部分必须相同，这限制了其使用范围。
如果产品的内部变化复杂，该模式会增加很多的建造者类。

建造者（Builder）模式和工厂模式的关注点不同：建造者模式注重零部件的组装过程，而工厂方法模式更注重零部件的创建过程，但两者可以结合使用。

4.2.3 装饰模式

说明：这个设计模式在HDFS源码用的次数较多。EditLogFileInputStream类中的nextOpImpl方法

```
public class DecoratePatternDemo {

    public static void main(String[] args) {
        SuperPerson superPerson = new SuperPerson(new Person());
        superPerson.superEat();

    }

    public static class Person{
        public void eat(){
            System.out.println("吃饭");
        }
    }

    public static class SuperPerson{
        private Person person;

        public SuperPerson(Person person) {
            this.person = person;
        }

        public void superEat(){
            System.out.println("来根烟把");
            person.eat();
            System.out.println("甜点");

        }
    }
}
```

总结：

介绍其适用的应用场景，装饰模式通常在以下几种情况使用。
当需要给一个现有类添加附加职责，而又不能采用生成子类的方法进行扩充时。例如，该类被隐藏或者该类是终极类或者采用继承方式会产生大量的子类。
当需要通过对现有的一组基本功能进行排列组合而产生非常多的功能时，采用继承关系很难实现，而采用装饰模式却很好实现。
当对象的功能要求可以动态地添加，也可以再动态地撤销时。

装饰模式在 Java 语言中的最著名的应用莫过于 Java I/O 标准库的设计了。例如，InputStream 的子类 FilterInputStream，OutputStream 的子类 FilterOutputStream，Reader 的子类 BufferedReader 以及 FilterReader，还有 Writer 的子类 BufferedWriter、FilterWriter 以及 PrintWriter 等，它们都是抽象装饰类。

4.2.4 组合设计模式

说明:其实在HDFS中这个设计模式使用得不是很明显，但是我们可以扩展一下，因为个人认为HDFS元数据的数据结构适合用这个设计模式。

如果不用设计模式:

```
/**
 *
 * 处理树状结构的数据：
 *
 * 1）权限
 * 2）主部门：
 *          子部门1：
 *                叶子部门1
 *                叶子部门2
 *          子部门2：
 *                叶子部门3
 *
 *
 */
public class WhihoutPatternDemo {
    public static void main(String[] args) {
        Department coreDep = new Department("主部门");

        Department subDep1 = new Department("子部门1");
        Department subDep2 = new Department("子部门2");

        Department leafDep1 = new Department("叶子部门1");
        Department leafDep2 = new Department("叶子部门2");
        Department leafDep3 = new Department("叶子部门3");

        subDep1.child.add(leafDep1);
        subDep1.child.add(leafDep2);

        subDep2.child.add(leafDep3);

        coreDep.child.add(subDep1);
        coreDep.child.add(subDep2);
        //需求 删除主部门
```

```
        if(coreDep.child.size() > 0){
            for(Department sub:coreDep.child){
                if(sub.child.size() > 0){
                    for (Department leaf:sub.child){
                         leaf.remove();
                    }

                }
                sub.remove();

            }
        }
        coreDep.remove();

    }



    public static class Department{
        private String name;
        private List<Department> child =new ArrayList<Department>();

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public List<Department> getChild() {
            return child;
        }

        public void setChild(List<Department> child) {
            this.child = child;
        }

        public Department(String name) {
            this.name = name;
        }
        public void remove(){
            System.out.println("删除"+name);
        }
    }


}
```

使用设计模式：

```
public class CompositePatternDemo {
    public static void main(String[] args) {
        Department coreDep = new Department("主部门");

        Department subDep1 = new Department("子部门1");
```

```
        Department subDep2 = new Department("子部门2");

        Department leafDep1 = new Department("叶子部门1");
        Department leafDep2 = new Department("叶子部门2");
        Department leafDep3 = new Department("叶子部门3");

        subDep1.child.add(leafDep1);
        subDep1.child.add(leafDep2);

        subDep2.child.add(leafDep3);

        coreDep.child.add(subDep1);
        coreDep.child.add(subDep2);

        //需求一： 删除主部门
        coreDep.remove();
        //需求二：修改名字 转转

    }

    public static class Department{
        private String name;
        private List<Department> child =new ArrayList<Department>();

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public List<Department> getChild() {
            return child;
        }

        public void setChild(List<Department> child) {
            this.child = child;
        }

        public Department(String name) {
            this.name = name;
        }

        public void remove(){
            if(this.child.size() > 0){
                for(Department department:this.child){
                    department.remove();
                }
            }
            System.out.println("删除部门"+ name);
        }


    }


}
```

总结：

组合模式的主要优点有：
组合模式使得客户端代码可以一致地处理单个对象和组合对象，无须关心自己处理的是单个对象，还是组合对象，这简化了客户端代码；
更容易在组合体内加入新的对象，客户端不会因为加入了新的对象而更改源代码，满足“开闭原则”；

其主要缺点是：
设计较复杂，客户端需要花更多时间理清类之间的层次关系；
不容易限制容器中的构件；
不容易用继承的方法来增加构件的新功能；

五、扩展延伸（5分钟）

1. JUC的知识
2. 多线程，并发，锁（wait，sleep，synchronized，notify，notifyAll等关键字）
3. JVM知识
4. 设计模式
5. 磁盘读写：NIO
6. 网络：socket，HadoopRPC，HTTP

六 、总结（5分钟）

6.1 阅读源码的方式：（道）

1. 掌握集群通信架构设计
2. 场景驱动（集群启动流程，元数据管理流程，读写数据流程）
3. 看类的注释
4. 连蒙带猜
5. 边看边画架构图，边写注释

6.2 需要掌握什么（道）

1. 分布式集群是如何通信的（Http，RPC，Soket）
2. 如何完成注册，心跳，监控
3. 分布式系统是如何容错的
4. 分布式系统是如何提升高并发的
5. 如何设计高质量的代码
6. 以后再看到其他分布式的系统，不再迷茫

6.3 需要搞定什么？（术）

1. HDFS的启动流程
2. HDFS元数据管理流程
3. HDFS的写数据流程
4. HDFS数据管道建立流程

6.4 需要反思什么？

1. 是否缺设计模式的知识
2. 是否缺多线程的知识
3. 是否缺IO的知识(NIO)
4. 是否缺JVM的知识
5. 是否缺计算机网络的知识

七、作业

1. 修改自己手里的hadoop-2.7.0的源码（不需要交）
2. 让大家画一遍我们的HDFS的源码的流程图。（需要交）

八、互动